A PROCESSOR INTERFACE MODEL FOR FAST SYSTEM SIMULATIONS

Gary Bolotin

Jet Propulsion **Laboratory**
California **Institute** of **Technology**
4800 Oak Grove **Drive**
Pasadena, [A **91109**
bolotin@telerobotics.jpl.nasa.gov

Abstract

This paper presents **an alternative** technique for performing **board** level simulations of designs involving **Processors. This technique** requires only **a** standard "C" **compiler** and **a few simulation** library functions to perform accurate **board** level simulations. **This method** is currently **being** used **to simulate the IBM 1750A based GVSC** inter-Subnsscn~bly **Bus** (1 **SB) and ASICs that are being developed** for **the Cassini** spacecraft. **This** meth()(I is superior **to** the conventional processor modeling **techniques,** which involve the use of **detailed hardware modeling** or **extensive behavioral models.**

1. Introduction

When developing ASICs that interface to Processors, a board level simulation is required to show that the ASICs under development will function properly at the system level. This functionality is usually verified by means of a system level simulation. This simulation} should involve all devices that interact with the ASICs under development. Simulat ions of this complexity arc traditionally long and cumbersome. Several complex and computationally intensive simulation models need to be used. As an example, system level simulations involving a processor require a model for it along with it's associated memory and I/O devices. T 'hc processor model is usually one of the most complex models required. Simulations of the processor alone can be quilt intensive and lime consuming. This paper presents an alternative way of modeling the processor and associated memory devices.

The paper will start by discussing the various techniques available for simulating processors. An alternative method for modeling the processor will then be introduced. "J his method will be illustrated by means of a simple example. This methodology can be used for modeling arbitrarily complex processors, and has been used at JPL to model the IBM 175(JA based GVSC Engineering Flight Computer (EFC) Inter-Subassembly Bus (ISB) used on the Cassini spacecraft,

2. Conventional **Processor** Modeling **Techniques**

Processors can be modeled using several different methods. The two most common methods are listed below. The third method, the processor interface model, the subject of this paper, will be discused in detail in an upcoming section.

1). Hardware model
2). Behavioral model (software - VI IDL)
3). The processor interface model.

2,1 **Hardware** Modeling **Techinque**

The hardware modeling technique makes usc of actual device hardware to perform the desired simulation. This technique can vary from building the system entirely in hardware, to using a hardware modeling library, similar to the Mentor H M L. [1] Both of these methods require the actual processor to perform the simulation. When building tge system entirely in hardware, FPGAs and PI Ds can be used to implement ASICs that have not been fabricated. Software is then written and executed to test for proper operation.

The hardware modeling 1 ibrary works by taking the output of the system simulation and applying it to an actual processor. The processor is thco clocked, and the. resulting sampled processor outputs, arc then applied as inputs 10 tile system simulation. The simulation is then clocked, and so on. Although this technique can be quite accurate, it can be very expensi vc in terms of resources and time.

2.2 Behavioral Modeling Techinque

The software modeling method requires the development of a software model which will simulate the processor. This can range from a high level description to a full gate level model. Software models can be developed using a hardware modeling language such as V}] Dl., or lhcy can be produced using a language native 10 the simulator. Software models arc typically required to model the Processor in every mode of operation, e.g. instruction fetch, execute, and prefetching of instructions. All internal registers and interactions may need to be modeled. Some processors may bc modeled all the way down 10 lilt gate level. The software that needs to bc developed for a software model can be quite extensive. A complicated software model might take. up a major portion of the simulators memory space and may slow down the simulation significantly.

The two software techniques just described, all model the processor down at least to the register level. In order to function properly both of lilt above methods rely on native code being presented to the processor model.

For example, assuming onc is simulating an 8086 based system cm a simulator running on a SUN workstation, onc would need to present 8086 native code. to the processor within the simulation. This code would control the operation of the processor and thus the verification of the system design. An 8086 cross compiler that runs on the SUN workstat ion is required to produce this code, The 8086 test software would be compiled and linked on the SUN workstation.

'I 'he. resulting executable image would then be loaded into the simulator. This can bc done, by initializing a ROM model with such code and telling the processor to begin execution from the ROM, a very time consuming process. In addition the processor will be repeatedly fetching code from memory. An interact ion which when shown to work once in the simulation, need not be repeat ed.

Another drawback is that once the processor fetches data of an unknown value, for whatever reason, the entire. simulation is like.iy to bc corrupted. Using this method it is also hard to detect errors when they occur. The results could be checked by hand, or code for monitoring the results needs 10 be written. This would just further lengthen simulation lime.

3. Processor Interface Modeling Technique

Our model is based on the fact that when using a simulator to verify a system level design, the designer is concerned mainly with tile proper interaction of the processor, mcrnory, I/O devices, and ASIC/s under development. This technique models the processor at the interface to the outside world. A detailed processor model is not required. This technique makes usc of only a standard "C" compiler and simple simulation library functions to simulate proper processor behavior.

This method simulates processor behavior by "forcing" the pins of tile processor or processor bus in such a way that proper operation is simulated. The basic development flow is shown in Figure 1. Test vectors arc produced using a "C" language lest program. The "C" language test program is compiled, linked and executed on tire simulator's host computer. When executed, the. test program will produce a test vector file that is then applied to the simulator. Using simulator commands these vectors manipulate the pins of the processor to simulate its operation in the system. Using a simulator system call, the vectors can also check for proper operation. For example on a mad cycle, the, lest vectors can test that proper data has been read back. An error message can be printed if the data read is not what is expected. A log of cycles performed can be kept simply by printing to the simulator's list window.

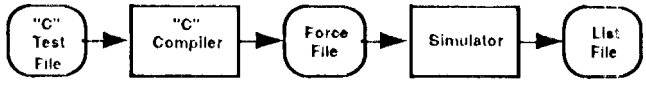

Figure 1. Test Vector Development Flow

3.1 Required Simulator System Functions

In order to check for proper simulation results it is required that the simulator have some way for the executing test vector program to have access to the value of nodes within the simulator. The Mentor Graphic Quicksim simulator can do this by means of the system call "$signal_value" and "$sim_time"[2]. The function $signal_value is called with a simulator net name and a simulation time. The routine will return the value of the net at the simulation time requested. The system value $sim_lime represents the. current simulation time.. These functions can be encapsulated in a simulator specific subroutine, check_output which compares a signal to an expected value, and will print an error message if values compared arc not equal. The simulator command "write line" is used to output to the simulator's list window. Ttrc routine, check output is called with the node that is to be compared arid the value that is expected.

```
check_output(node_name,data_expected)
char* string
unsigned int data_expected;
(
        printf("assign val ^($signal_value ('%s',
                $sim_time))\n",node_name);
        printf("if ^(val<> '%x' then\n",data_expected);
        printf(" write 1 inc 'Compare Failure' ^$sim_time
                'Unexpected value' 'vai ' Expected '
                ^%x 'on signal %s'\n", data_expected,
                node_name);
        printf("end if");
```

}

Figure 2. Check Output Routine

3.2 Example

As an example, suppose wc want to simulate a processor with a very simple asynchronous bus protocol as illustrated in figure 3, 4 and 5, and described as follows. For the purpose of this example the Mentor Graphics Quicksim simulator is to be assumed.

Write Cycle

1). Address and Data are put on to the bus.
2). The signal ADVn is asserted.
3). Wait for slave to assert DTACKn.
4). Deassert ADVn.
5). Remove Address and Data.
6). Wait for DTACKn to be deasserted.

Read Cycle

1). Address is put on to the bus.
2). The signal ADVn is asserted.
3). Wait for slave to assert DTACKn.
4). Sample Data lines.
5). Deassert ADVn.
6). Remove Address and Data.
7). Wait for DTACKn to be deasserted.

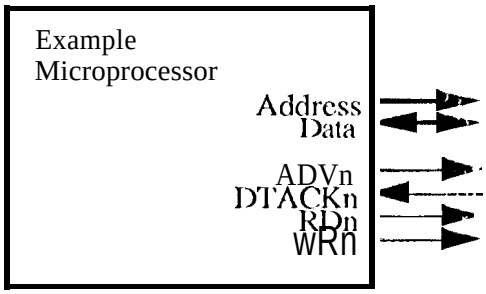

Figure 3. Example Processor

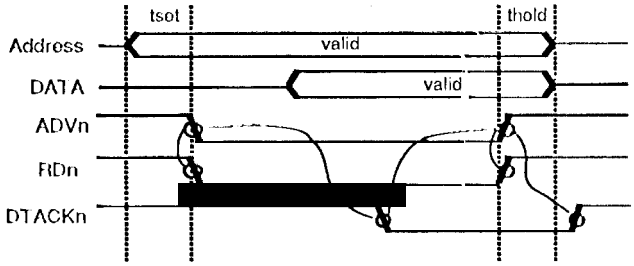

Figure 4. Read Cycle

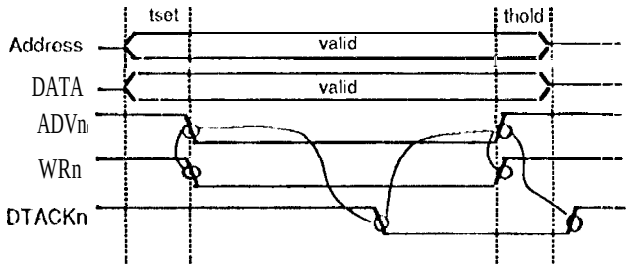

Figure 5. Write Cycle

3.2.1 Basic Routines

Two basic routines need to be written, one to simulate a read cycle and another one to simulate the write cycle. The read routine is called with the address of the data word that is to be read, and the expected value of this word. If the data read is different from the expected value, an error message is printed to the simulator's list window. This is handled by the routine check_output described above. The read routine is shown in figure 6.

```
read(address,data)
unsigned int address,data;
{
        printf("write line ' READ address:%x data
                %x\n",address,data_expected);
        printf(''forcc, ADDRES S %x\n",address);
        prin[f(''forcc I DATA %x\n",address);
        printf("run %d\n",TSET)
        ])rirllr(''forcc RDn 0\n");
        pril]tf(''forcc ADVn 0\n");
        prinlr(''brcak DTACKn 0\n");
        chink. output(DATA,data_expected);
        prinlr(''forcc RDn 1\n");
        printf'(''forcc ADVn lb");
        printf("run %d\n",THOLD)
        ])rilltl_(''forgc.t force ADDRESS \n");
}
```

Figure 6, Read Routine

The write routine is called with the address of the data word that is to be written, and it's value. When executed, a simulated write operation is performed. The write routine is shown in figure 7.

```
wri te(address,data)
unsigned int address,data;
{
        printf("write line' WRITE address: %x data
                %x\n",address,data);
        printf("force ADDRESS %x\n",address);
        prin!f("run %d\n",TSET)
        prin'.f(''forcc. AI DVn 0\n");
        prin!f(''forcc WRn 0\n");
        prinlr(''brcak I DTACKn 0\n");
        cl neck_output (I DATA,data_expected);
        printl_(''forcc WRn 1\n");
```

```
    pr'iatf(''forcc ADVn1\n");
    printf(''run %*'',T'}101 .D)
    prin[f(''forgct force ADDRESS\n);
    printf("forget force DATA\n);
)
```

Figure 6. Write Routine

3.2.2 Main Routine

We can now use the above routines to simulate the processor interfacing with the rest of the system. Instruction fetches can be simulated by reading from ROM or RAM memory spaces. ASIC and 1/0 functionality can bc tested by writing to ASIC or 1/0 device registers and reading back the, expected results,

As an example., wc can usc the routines just described, to test a block of ASIC registers that arc bcth readable and writeable. We can write a simple. loop to perform this function as follows.

```
. . . . .
for (addr = START_BLOCK; addr <=
                    STOP_BLOCK; addr++ )
        write(addr,PATTERN);
        read(addr,PATTERN);
}
. . . . .
```

Figure 8. Simple Test Loop

The test loop shown above would write the value PATTERN to every register from START_BLOCK to STOP_BLOCK. After the write operation is completed, proper operation is check by reading the register back using the read routine. An ongoing log of results is kept in the simulator's list window. Any errors encountered during the simulation arc also recorded there. Figure 9 shows and example from the simulator's list window.

```
. . . . .
WRITE address: 78F5 data: F34E
READ address:78F5 data: P3411
WRITE address: 78F6 data: F34E
READ address:78F6 data: F34E

WIW1'11 address:78F7 data: F34E
READ address:78F7 data: F34E
Compare Failure Unexpected Value: F34F
                Expected:F34E !!!

. . . . .
```

Figure 9. Example Simulator's List Window

3.3 Discussion

'1 'he benefits of the described method are portabil ity, simplicity anti efficiency. The same routine that is developed to test the ASICs and 1/0 devices in the simulated environment can be also used to test the ASICs and I/O devices in the rcai system afte: the ASICs arc fabricated. 'i 'his is made possible because the. "C" language is portable from system to system. 1 f on the other hand, the test vectors were written in a sim ulator specific language, the test vectors developed will have little use outside the. simulator. 'l'his technique frees the user from developing code native to the target processor thus allowing the user to concentrate on the task at hand; verifying that the proper interaction of the processor, ASICs and other dc.vices within the system. The simple nature, of the modeling method makes efficient use of simulation resources.

4. Conclusion

Wc have presented a simple, elegant and inexpensive technique for performing system simulations. The technique is currently being used with great success at JPl. to model the IBM 1750A based GVSC Engineering Flight Computer (EFC) Inter-Subassembly Bus (1 SB) used on the Cassini spacecraft. .

5. References

[1] Mentor Graphics, Hardware Modelling Library !&Ls-._Miul.ua!, Version 6.0, 1990.

[2] Mentor Graphics, System Calls and Library Functions, Version 6.0, 1990.

6. Acknowledgm ent

This research described in this paper was carried out by the Jet l'repulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space, Administration.